MythOS – A Micro-Kernel Threads Operating System*

Frank Mueller, Viresh Rustagi and Theodore P. Baker
Department of Computer Science, 4019
Florida State University
Tallahassee, Florida 32306-4019
phone: (904) 644-3441, *e-mail: mueller@cs.fsu.edu*

Abstract

MythOS (Micro-kernel THreads Operating System) is an experimental operating system for embedded systems. The system kernel is a first implementation of the POSIX "Minimal Real-Time System Profile". It is based on prior work of a library implementation of Pthreads (POSIX threads). The system is fully preemptive. It supports multi-threading within a single process environment with shared kernel and user space. It exhibits remarkable timing predictability intended for hard real-time requirements. This is achieved by a careful design of only few device drivers. The system has been implemented and tested on the SPARC VME architecture. Also presented is a fast context switching algorithm for the SPARC which outperforms the context switch under SunOS 4.x and matches the performance under Solaris 2.3. Furthermore, an implementation-defined extension of Pthreads for deadline scheduling is presented. Overall, the system exhibits slightly faster performance than SunOS 4.x and is considerably more predictable in its timing behavior.

1 Introduction

Lately, micro kernels have often been chosen for the design of real-time operating systems [8, 6, 2]. Yet, most of the systems implement a large subset of the UNIX system interface. Additional functionality of a system often degrades the timing predictability and may also contribute to more costly kernel calls. Many modern operating systems acknowledge the demand for faster, simpler programming models. One result is the model of separate threads of control within a process. A version of model is in the process of becoming the IEEE POSIX 1003.4a Threads Extension Standard, for short *Pthreads*. The draft 6 version of Pthreads has been implemented as the FSU Pthreads library for the SPARC architecture [11]. The implementation was reported to perform faster than kernel-based commercial threads implementations.

This paper describes the micro-kernel MythOS. MythOS implements a major part of the "Minimal Realtime System Profile" of IEEE POSIX 1003.13 draft [9] by integrating the FSU Pthreads library with a micro kernel for the SPARC architecture. The minimal profile is intended for "embedded systems dedicated to unattended control of one or more special I/O devices" and does not support memory management. Our system is being gradually extended to eventually support the entire profile. The minimality of the system results in some performance benefits and better timing predictability than larger kernels. For example, sharing user and kernel space preserves the performance benefits for the micro kernel that were exhibited by the library implementation. A performance evaluation between the library implementation and the kernel implementation can be found in [3]. This work showed that MythOS performs 1 to 3 times faster than the Pthreads library on top of UNIX, depending on the kernel calls. Measurements also showed that MythOS exhibits very predictable timing behavior for repeated program executions. Under UNIX, on the other hand, the timing predictability sometimes dropped drastically due to kernel activities.

One of the motivations for the implementation of MythOS was to provide an embedded test bed for GNAT (Gnu Ada 9X translator) [15] in conjunction with our Pthreads-based GNARL (Gnu Ada 9x Run-Time Library) [7]. Another motivation was to provide a micro kernel for a modern architecture whose source code is publicly available and can be modified or extended.

The paper is structured as follows. Section 2 gives an overview of the design of the MythOS micro kernel. Section 3 describes an implementation of a fast

*This work was partially funded by the Ada Joint Program Office, through the U.S. Army CECOM and Telos Corp.

context switching algorithm for the SPARC. Section 4 introduces extensions to the Pthreads standard for deadline scheduling support. Section 5 discusses future work. Section 6 reviews previous work. Section 7 summarizes the paper. Finally, the appendix presents an implementation of the fast context switching algorithm.

2 MythOS Micro Kernel Design

In the following, the design of the underlying micro kernel of MythOS is described. The basic design of the high-level kernel functionality is identical to the design and implementation of the FSU Pthreads library [11]. The careful design of this preemptive library was intentionally close to the high-level design of a micro kernel. It includes a monolithic monitor for kernel operations which is entered by simply setting a flag. If scheduling actions are performed the kernel is left through the dispatcher; otherwise is suffices to clear the kernel flag to leave the kernel. A more detailed description of the high-level design can be found in [11] and will not be reiterated here.

This section details some of the low-level design issues of the micro-kernel. It breaks with some traditions of conventional kernel design in order to achieve better performance and timing predictability.

- User space and kernel space are not distinguished. There is no memory protection for the kernel data structures. Thus, kernel traps which change the memory protection at trap entry and exit become obsolete. Kernel calls can be made directly via regular function calls.
- Interrupt handlers are restricted to the bare minimum: a timer, serial I/O, ethernet (not yet implemented), and VME memory (not yet implemented).
- Each interrupt handler executes its own handler and transfers control to a common routine only for preemption.
- Preemption is supported through a common routine. The control can be transferred to this code from any interrupt handler. The design integrates the handling of process-level signals.
- Virtual memory management (paging) is not supported. Programs execute directly in "physical memory".

The sharing of user space and kernel space can be risky. Kernel data structures can easily be overwritten and there is generally no way to recover from such errors. On the other hand, consider a system with a protected kernel. Kernel calls are made through a common trap, the kernel data space is made accessible, and a specific kernel routine is selected via demultiplexing. In MythOS the kernel is entered directly (by setting a flag) and control is transferred to a specific routine via a regular function call which avoids the traditional demultiplexing of kernel calls. By sharing user and kernel space without protection MythOS still lowers the kernel call overhead under the assumption that the user code does not violate memory constraints.

A similar issue is posed by interrupt handlers. In traditional kernel design most device-triggered interrupts share a common entry and exit code for handling the interrupt. The entry and exit code handles page faults, addresses architecture-specific characteristics, and then demultiplexes the different interrupts by calling a specific device driver. The device drivers can thereby be mostly written in a higher level language instead of assembly. In MythOS each device-triggered interrupt has a separate interrupt handler. The basic device driver functionality (such as time keeping or character/block processing) is performed within the interrupt handler. Higher-level operations (such as scheduling) are performed by transferring control from the interrupt handler to a common preemption routine. This higher-level portion of the device drivers is written in C. Again, demultiplexing of interrupts is avoided. The most frequently executed small portion of a device driver is executed without the overhead of lengthy entry and exit code. Notice that this approach is adopted for certain frequently executed interrupts in traditional kernels, *e.g.* for the timer interrupt.

The common preemption routine replaces in effect the signal trampoline of traditional UNIX implementation, yet at the level of interrupts. The semantics of the preemption routine depends on whether user code or kernel code was interrupted.

If user code of a thread is interrupted and the interrupt handler determines that a scheduling action needs to be performed, then control is transferred to the preemption routine. This routine enters the kernel (by setting a flag) and creates a new stack frame on top of the interrupt handler. It then calls a routine which performs the scheduling action, followed by a call to the dispatcher. The current thread may lose the processor at this point. When the thread regains control of the processor it will resume its execution by returning from the interrupt handler.

If kernel code of a thread is interrupted and the interrupt handler transferred control to the preemption routine a process level signal corresponding to the interrupt is raised. The scheduling effect of the interrupt is deferred, *i.e.* preemption does not occur. The execution will then resume by returning from the interrupt handler but will still be inside the kernel code. When the kernel is left the process signal mask is examined and pending signals are handled before the dispatcher attempts to switch contexts.

The efficient handling of interrupts in the described manner is certainly simplified by the absence of paging. A common entry and exit code for interrupt handlers becomes almost unavoidable when a kernel supports virtual memory management. While the programming model is greatly simplified at a high level by virtual memory support it may introduce timing distortion at a lower level. The overhead of a page fault can hardly be predicted in a preemptive kernel which may replace a user page with a kernel page at arbitrary points. This can be circumvented by locking pages in memory. Yet, in the single process environment of MythOS memory is shared between threads without any protection. A thread may overwrite the memory of one or even all other threads without any chance of recovery, even if the kernel space was protected. Therefore, MythOS does not support virtual memory management. In addition, it may be argued that the physical memory size of today's computers is sufficient to execute most programs in embedded environments.

While implementing MythOS on the SPARC a lot of time was spent on debugging the preemption routine. It is our experience that the register window model of the SPARC can apparently be understood fairly easily at a high level. However, when code efficiency requires direct manipulation of the CWP (current window pointer) and WIM (window invalid mask) registers mistakes are often made during coding due to some extreme cases which were not considered. A RISC architecture with a flat register file would have facilitated the job considerably.

3 A Fast Context Switch for the SPARC

One of the frequently executed portions of operating system code is the context switching code, especially for real-time operating systems. During a context switch on the SPARC all register windows of the current thread are flushed onto the thread's stack before one window will be loaded with the top frame of the new thread. Flushing register windows requires access to the special registers CWP (current window pointer) and WIM (window invalid mask). This access is only granted in supervisor mode. Thus, a dedicated kernel trap is used by operating systems on the SPARC to perform this operation.

The implementation of the context switch under SunOS 4.x simply flushes all register windows of the processor. Yet, a thread may not use all register windows at any point of time. The number of windows used depends on the dynamic nesting depth of calls between context switches. In a real-time operating system with frequent context switches it is less likely that a thread uses all of the windows. It would therefore be useful to implement the context switch such that only the windows currently in use are flushed to memory. This information is readily available when CWP and WIM are compared.

We have designed and implemented a window flush algorithm for the SPARC which only flushes used register windows. This algorithm is presented in Figure 3 in the appendix. The algorithm is essential to speed up context switching on the SPARC. We evaluated the performance of this algorithm under MythOS by comparing it with the register window flushing under SunOS 4.0.3e on a SPARCEngine 1E VME board clocked at 20MHz. We also interpolated these results with the performance exhibited under SunOS 4.1.3, Solaris 2.2, and Solaris 2.3 on a SPARCclassic clocked at 50MHz. The experiment was conducted by using dual-loop timing and averaging 1000 samples of `setjmp()/longjmp()` pairs with a call depth between the calls varying from 0 to 6. The `longjmp()` call performs a trap to flush register windows. The resulting measurements are shown in Figure 1.

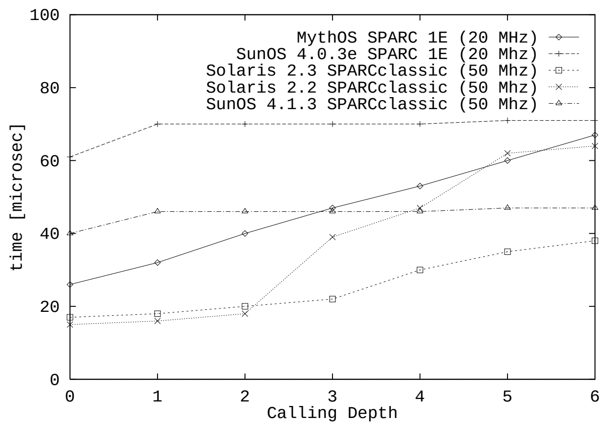

Figure 1: Context Switch Performance on the SPARC

The register window flushing performance under SunOS 4.x is essentially constant. This is consistent with the observation that all register windows are saved, regardless of the actual usage. The MythOS implementation shows a steady increase in the performance linear to the calling depth. This is due to the selective flushing of only the used windows and may result in a speedup factor of 1.1 to 2.3 (on a SPARCEngine 1E at 20 MHz). The performance under Solaris 2.3 also increases (almost) linearly with the calling depth. The speedup factor of Solaris 2.3 over SunOS 4.1.3 is 1.2 to 2.3 (on a SPARCclassic at 50 MHz). The interpolation via speedup factors seems to indicate that the algorithm used under Solaris 2.3 also

```
int pthread_attr_getstarttime_np(pthread_attr_t *attr, struct timespec *tp);
int pthread_attr_setstarttime_np(pthread_attr_t *attr, struct timespec *tp);
int pthread_attr_getdeadline_np (pthread_attr_t *attr, struct timespec *tp);
int pthread_attr_setdeadline_np (pthread_attr_t *attr, struct timespec *tp,
                                 void *(*user_handler)(void *));
int pthread_attr_getperiod_np   (pthread_attr_t *attr, struct timespec *tp);
int pthread_attr_setperiod_np   (pthread_attr_t *attr, struct timespec *tp,
                                 void *(*user_handler)(void *));
```

Figure 2: Implementation-Defined C Interface for Deadline Scheduling

saves only the windows currently in use and may in fact be very similar to our implementation. The performance under Solaris 2.2 may be due to an earlier version of the algorithm which sometimes performed better, sometimes worse than saving all register windows.

It should be remarked that these experiments exhaust the savings of the improved context switch. The SPARC architectures tested only implement 7 register windows. Once the calling depth matches or exceeds the number of register windows all windows need to be saved during a context switch. This situation is approximated by a calling depth of 6. Of course, each call past a nesting depth of 6 causes a window overflow trap which flushes a single window to memory.

4 Real-Time Support

The POSIX threads standard specifies a general interface with support for real-time systems. MythOS implements strict priority scheduling and round-robin priority scheduling with preemption. It supports the regular mutex locking protocol and a priority ceiling protocol which can be used to bound priority inversion. The ceiling protocol implemented is called SRP (Stack Resource Protocol [4]) and can reduce the amount of context switches compared to conventional ceiling protocols.

Yet, the POSIX threads standard does not specify any interface for deadline scheduling of a thread. We support an implementation-defined extension of the POSIX threads interface orthogonal to the overall interface design. The scheduling attributes of a thread are extended to allow the specification of an absolute start time, an absolute deadline, and a relative period (see Figure 2). These attributes have to be set before passed as an argument for thread creation. They cannot be changed dynamically while the thread is already running but the value can be inquired at any time.

The semantics for these deadline-scheduled threads is as follows: A thread with a start time begins to execute no earlier than its start time. A thread with a period (and a required start time) is restarted when when its prior start time plus its period is reached. If the thread does not complete its execution within its period, a user-defined handler is invoked at the end of its period before it is restarted. For a thread with a deadline (and a required start time) this user handler is invoked when its deadline is reached if it has not yet completed its execution. The user handler executes as a signal handler. It is commonly used to lower the priority of a thread or to terminate the thread via a call to `pthread_exit()`.

Periodic threads are implemented by creating the thread only once and reinvoking the thread function repeatedly. This is accomplished inside a wrapper which calls the thread function. The wrapper requests a thread-specific timeout signal for the end of the period and calls the thread function. If the execution completes within the period the thread returns from the thread function, the timeout signal request is cancelled, and the thread suspends until the period expires. Then, this process is reiterated. If the thread does not complete within its period, the user handler is invoked asynchronously. Since MythOS system calls are async safe it is possible to make kernel calls in this handler. For example, if the thread is terminated via `pthread_exit()`, then the cleanup handlers are called, yet the control is transferred back to the wrapper which will set up the next invocation. Deadlines are implemented using the same timeout request but without reiterating the invocation of the thread's function.

The cleanup handlers executed during thread cancellation can be used to wrap up a computation which did not complete in time. This way, held mutexes can be released or intermediate results can be used for imprecise computations [5]. The overhead of cleanup handlers and the cancellation routine has to be added to the estimated execution time of a thread when performing schedulability analysis. The schedulability analysis is currently assumed to be performed off-line.

5 Bounding Device Activities

One of the problems for real-time system stems from the difficulty to predict the frequency of interrupts. While the timer interrupt occurs in regular intervals the same cannot be said about other device drivers. For example, if too many I/O interrupts occur during a short interval a thread's deadline may be missed. This problem can be dealt with by the aperiodic server model [10] adapted for interrupt handlers. MythOS is designed to bound the number of interrupts during a certain time interval on user request. If the number of interrupts exceeds the bound further interrupts are not processes. For example, characters received from a serial interface are simply ignored after a certain number of characters have been read during an interval. The loss of information is traded for the completion of a thread's execution. The bound for a device has to be determined during schedulability analysis and can be set during program execution. This provides an effective method to control the responsiveness of the system while preserving deadlines.

6 Future Work

The deadline-scheduling support is intended to be enhanced by real-time accounting at the kernel level. The ethernet driver and the VME memory drivers have not yet been implemented. We are also planning to port MythOS to a Force 3E VME board with a MicroSPARC I processor and regular SPARC workstations. At some point, we would like to attempt a multiprocessor implementation. This would require finer locking mechanisms at the level of each thread control block rather than just one monolithic monitor.

7 Related Work

Alfieri [1] discussed a kernel-based implementation of POSIX threads. He also observed that conventional kernel traps are too inefficient. His solution uses a general-purpose trapping mechanism, yet the overhead of the reduced kernel trap entries and exits is much less than in conventional systems.

Several operating systems use threads, mostly as "multiplexed" libraries with two layers of scheduling. For example, Sun implements scheduling at the library level for threads and at the kernel level for light-weight processes [12, 13]. A number of real-time systems provides support for deadline scheduling of threads, including LynxOS [6], Chorus [2], Real-Time Mach [14]. All of these systems support a large subset of a UNIX system interface whereas MythOS provides a small kernel for multi-threaded embedded applications.

8 Conclusion

MythOS provides a first implementation of the POSIX "Minimal Realtime System Profile". Calls to the operating system are relatively fast due to the shared user and kernel data. The timing characteristics of the system are more predictable than in comparable systems due to the careful design of only few device drivers. The activities of device drivers can be controlled using an aperiodic server model. The design of operating system calls and interrupt handlers makes common entry and exit code with the associated demultiplexing of system services obsolete which contributes to the performance. The MythOS kernel is relatively small with about 40kB in code size. The system provides an implementation-defined extension of the Pthreads standard to support deadline scheduling. Thus, MythOS can be used as an experimental real-time operating system for embedded systems.

Availability

The MythOS source code will soon be available on ftp. The source code of the FSU Pthreads library is already available under Gnu copyright from `ftp.cs.fsu.edu` (128.186.121.27) in `/pub/PART/pthreads.tar.Z` – other material such as related publications can be found in the same directory.

References

[1] R. Alfieri. An efficient kernel-based implementation of posix threads. In *USENIX Conference*, Summer 1994.

[2] Francois Armand, Frederic Herrmann, Jim Lipkis, and Mark Rozier. Multi-threaded processes in CHORUS/MIX. In *EEUG Conference*, pages 1–13, Spring 1990.

[3] T. P. Baker, F. Mueller, and Viresh Rustagi. Experience with a prototype of the POSIX "minimal realtime system profile". In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 12–16, 1994.

[4] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[5] Jen-Yau Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs using imprecise results. Technical Report UIUCDCS-R-87-1307, Departement of Computer Science, University of Illinois, November 1987.

[6] Bill O. Gallmeister and Chris Lanier. Early experience with POSIX 1003.4 and POSIX 1003.4a. In *IEEE Symposium on Real-Time Systems*, pages 190–198, December 1991.

[7] E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.

[8] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, April 1992.

[9] IEEE. *Standardized Application Environment Profile—Posix Realtime Application Support (AEP) (Draft 5)*, February 1992. P1003.13/D5.

[10] J. Lehoczky, L. Sha, and J. Stosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *IEEE Symposium on Real-Time Systems*, pages 261–270, December 1987.

[11] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[12] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *USENIX Conference*, pages 65–80, Winter 1991.

[13] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX Conference*, pages 1–10, Summer 1992.

[14] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-Time MACH: towards a predictable real-time system. In *USENIX MACH Workshop*, October 1990.

[15] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.

```
! Copyright (C) 1994, the Florida State University
! This code can be used and changed free or charge, wihtout any waranty,
! as long as this copyright notice is included.

#define LOCORE
#include <sun4e/allregs.h>
#include <sun4e/psl.h>
#include <sun4e/asm_linkage.h>

!       void flush_windows()
!
!       Flushes all valid windows on stack and sets the WIM such that
!       next restore shall cause an underflow and hence the registers shall
!       be read from stack.
!       Assumes that the following values in registers:
! %l1 = %pc
! %l2 = %npc
! %l7 = %psr

        .seg        "text"
        .global     _flush_windows
_flush_windows:
        mov         %wim, %l3           ! cur_wim = get_wim();
        and         %l7, PSR_CWP, %l4   ! next_wim = 1 << (cur_psr & PSR_CWP);
        mov         1, %l5
        sll         %l5, %l4, %l5

        sll         %l5, 2, %l4         ! new_wim = next_wim << 2;
        srl         %l5, NWINDOW-2, %l5 ! next_wim >>= NWINDOW - 2;
        or          %l5, %l4, %l5       ! next_wim |= new_wim;

        andcc       %l5, %l3, %g0       ! (next_wim & cur_wim) != 0
        bnz,a       flush_ret           ! Yes
        mov         %l7, %psr           ! Delay: Restore psr iff jump

        mov         %g1, %l0
        mov         %g2, %l6
        mov         %l5, %g1
        mov         %g5, %l5
        mov         %g1, %g5
        mov         %l3, %g2
        mov         %g4, %l3
        mov         %l7, %g4            ! old_psr = cur_psr;
        mov         %g3, %l4

        restore                         ! from trap to caller frame
        restore                         ! goto previous window
3:
        SAVE_WINDOW(%sp)
        sll         %g1, 1, %g3         ! new_wim = next_wim << 1;
        srl         %g1, NWINDOW-1, %g1 ! next_wim >>= NWINDOW - 1;
        or          %g1, %g3, %g1       ! next_wim |= new_wim;
        andcc       %g1, %g2, %g0       ! (next_wim & cur_wim) == 0
        bz,a        3b                  ! Yes - goto SAVE WINDOW
        restore                         ! Delay: goto previous window iff branch

        mov         %g4, %psr           ! Restore psr
        mov         %g5, %wim           ! set wim
        nop                             ! Delay:
        nop                             ! Delay:

        mov         %l0, %g1
        mov         %l6, %g2
        mov         %l4, %g3
        mov         %l3, %g4
        mov         %l5, %g5
flush_ret:
        jmpl        %l2, %g0            ! Delay: ret to instr after ta call(nPC)
        rett        %l2+4               ! Delay: Set nPC to old nPC+4
```

Figure 3: Fast Register Window Flushing Algorithm for the SPARC